

Backup and Restore

- [Overview](#)
- [Usage](#)
 - [Backup](#)
 - [Restore](#)
- [Configurations](#)
- [Backup Format](#)
- [Filesystem Backup](#)
- [LevelDB Backup](#)
 - [Backup Strategies](#)

Overview

The Fedora 4 Backup capability allows a user, such as the repository manager, make a REST call to have the repository binaries and metadata exported to the local file system. Inversely, the Restore capability allows a user to make a REST call to have the repository restored from the contents of a previous Backup operation. In addition, with the default configuration, files are stored on disk named according to their SHA1 digest, so a filesystem backup approach can also be used.

Design Considerations

Historically, Fedora fulfilled its promise of durability by choosing transparent forms of persistence (e.g. human-readable XML) and using them in ways that systems outside the repository could readily penetrate if needed. Transparency in support of durability is as valid a principle as ever, but there is a weakness to it: transparent forms of persistence are not performant. What's more, many users didn't particularly care for that principle, but they were still stuck paying the performance costs associated with it. So in Fedora 4, we shifted responsibility for transparent persistence away from the core repository software. If you'd like to maintain some simple, human-readable form of your repository, that's fine, but you need to support that with [an integration around the core](#). The form of persistence used by the core repository component itself is not meant to be manipulated directly by a human except in the most unusual situations, it's meant instead for use by the software to provide speedy service at the repository's API. You might compare this to the use of database software. You don't expect to directly manipulate database indexes, and if you are concerned for the durability of your data in the database, you take backups in a transparent format and use `_those_` to ensure durability.

An analogy: you may expect your bank to provide downloadable images for any checks you write, but you don't expect them to use those images to run their accounting software.

Usage

Backup

If a POST body specifying a writeable directory (local to Fedora 4 server) is not included in the request, the backup will be written to the system temp directory.

Perform a backup of a running Fedora 4 repository

Request

```
POST /rest/fcr:backup
> optional POST body
```

Response

On success

- HTTP/1.1 200 OK
- Path where the backup was written

Response body

- Absolute path of local backup directory

Restore

Note: Restoring a backup replaces the repository content with the contents of the backup, so any data in the repository will be lost.

Perform a restore of a running Fedora 4 repository

Request

```
POST /rest/fcr:restore
```

```
> with POST body
```

A POST body containing the full path to a previous backup.

Response

On success

- HTTP/1.1 204 No Content

Configurations

The following configurations have been successfully tested with the Backup and Restore functionality

- Non-clustered Fedora, using Infinispan cache backed by LevelDB ([config](#))

Backup Format

Regardless of the repository configuration, the output of the backup process creates resources of the same format. Further details on backup contents and the underlying implementation can be found in ModeShape's [documentation](#).

The backup directory will contain

- 'binaries' directory that contains the repository "content" binaries stored in a pair-tree like structure. The filename of the binary is the SHA-1 of the content with the extension '.bin'. The directory structure in which each binary is found is three levels deep based on the SHA-1.
 - For example, binary content in the repository with a SHA-1 of "5613537644c4d081c1dc3530fdb1a2fe843e570170d3d054", would look like

```
binaries
  44
    c4
      d0
        44c4d081c1dc3530fdb1a2fe843e570170d3d054.bin
```

- One or more "documents_00000n.bin.gz" files which contains a concatenated listing of the metadata for each of the repository objects in JSON format
 - For example

```

{ "metadata" :
  { "id" : "87a0a8c317f1e7/jcr:system/jcr:nodeTypes/nt:unstructured//undefined/1" ,
    "contentType" : "application/json" } ,
  "content" :
  { "key" : "87a0a8c317f1e7/jcr:system/jcr:nodeTypes/nt:unstructured//undefined/1" ,
    "parent" : "87a0a8c317f1e7/jcr:system/jcr:nodeTypes/nt:unstructured" ,
    "properties" :
    { "http://www.jcp.org/jcr/1.0" :
      { "primaryType" :
        { "$name" : "nt:propertyDefinition" } ,
        "onParentVersion" : "COPY" ,
        "multiple" : false ,
        "protected" : false ,
        "availableQueryOperators" :
        [ "jcr.operator.equal.to" ,
          "jcr.operator.greater.than" ,
          "jcr.operator.greater.than.or.equal.to" ,
          "jcr.operator.less.than" ,
          "jcr.operator.less.than.or.equal.to" ,
          "jcr.operator.like" ,
          "jcr.operator.not.equal.to" ] ,
        "requiredType" : "UNDEFINED" ,
        "mandatory" : false ,
        "autoCreated" : false }
      }
    }
  }
}

```

Filesystem Backup

By default, files larger than 4KB are stored on disk named after their SHA1 digest, in the directory `fcrepo.binary.directory`. (4KB is the default, but can be changed by updating the `minimumBinarySizeInBytes` property in [repository.json](#)). That is, a file with the SHA1 "a1b2c369563c0465ab82cdb2789d45ce1c3585b1" would be stored on disk in `/path/to/fcrepo4-data/fcrepo.binary.directory/a1/b2/c3/a1b2c369563c0465ab82cdb2789d45ce1c3585b1`. So files in the repository can be backed up backing up the directory `fcrepo.binary.directory`.

LevelDB Backup

LevelDB stores its data as flat files in the directory `fcrepo.ispn.repo.cache` of the `fcrepo` home. The `fcrepo` home directory can be backed up as a whole to create a snapshot of the repository with both the binaries and the metadata. Though, the `fcrepo.binary.directory` and `fcrepo.ispn.repo.cache` are the only directories necessary for backup. (See [ModeShape Artifacts Layout](#)). The backup can be created on a live repository without having to shutdown or restricting ingests to the repository. Though, it would be good idea to schedule the backups after any batch ingests, so that the newly ingested data is also included in the backup.

Backup Strategies

Here are a few strategies for backup:

WITH SHUTTING DOWN FEDORA (CONSISTENTLY RELIABLE BACKUPS)

STEPS:

1. Shutdown Fedora
2. Backup of FCREPO HOME (or just `fcrepo.binary.directory` and `fcrepo.ispn.repo.cache`)
3. Restart Fedora

WITH PAUSING WRITES TO FEDORA

STEPS:

1. Pause all updates to the repository.
 - a. Do not create, delete, or update OBJECTS or DATASTREAMS.
2. Wait for all previous update requests to be processed.

- a. For serialization of newly created objects to complete.
 - b. And, for leveldb background compaction to complete (usually in seconds), if the previous updates triggered a compaction.
3. Backup of FCREPO HOME (or just fcrepo.binary.directory and fcrepo.ispn.repo.cache)
 - a. Verify successful backup.
4. Continue with the updates.

HOT BACKUPS (LESS RELIABLE UNLESS VERIFIED)

1. Backup of FCREPO HOME (or just fcrepo.binary.directory and fcrepo.ispn.repo.cache)
 - a. Verify successful backup.

Verifying Backups:

1. Check leveldb cache store directory from the backup using a leveldb client.
 - a. Verify the leveldb opens.
 - b. Verify by iterating through the keys. (To expose any corruption)
2. Based on the flow of the background compaction process in the leveldb implementation ([1] and [2]), the manifest file is updated at the end of compaction, which is followed by the deletion of obsolete files. To verify successful backups, we can begin the backup with copying the manifest file followed by the rest of the files. And, at the end of the backup, the backed up manifest file can be compared with the current manifest. An unchanged manifest can be considered as a successful backup, and vice versa.

[1] https://github.com/google/leveldb/blob/master/db/db_impl.cc#L655

[2] https://github.com/google/leveldb/blob/master/db/version_set.cc#L811

The following script can be used to perform hot backup with verification (requires repair and verify scripts):

backup_leveldb.sh

```
#!/bin/bash

# Location of the fcrepo home directory
FCREPO_HOME=/var/lib/tomcat7/fcrepo4-data
# Destination Directory
BACKUP_TO=/home/vagrant/backup
# Backup exact (without the structural changes introduces by Python verification script).
# (Additional temporary storage is used, if true)
BACKUP_EXACT=true
# Max Backup Attempts on failure
ATTEMPTS=10

echo `date`" $0: fcrepo home dir: $FCREPO_HOME"
echo `date`" $0: backup dir: $BACKUP_TO"
echo `date`" $0: max attempts on failure: $ATTEMPTS"

if [ ! -d $BACKUP_TO ]; then
  mkdir $BACKUP_TO
fi

LEVELDB_DIR=fcrepo.ispn.repo.cache
DATA_DIR=dataFedoraRepository
backup_succeeded=false
attempts=$ATTEMPTS

echo `date`" $0: Backing up leveldb."
while [ $attempts -gt 0 ]; do
  MANIFEST_FILE=`ls $FCREPO_HOME/$LEVELDB_DIR/$DATA_DIR/MANIFEST-*`
  MANIFEST_MD5=`md5sum $MANIFEST_FILE`
  rm -rf $BACKUP_TO/$LEVELDB_DIR"-tmp"
  cp -r $FCREPO_HOME/$LEVELDB_DIR $BACKUP_TO/$LEVELDB_DIR"-tmp"
  copy_success=?
  MANIFEST_FILE_POST_BACKUP=`ls $FCREPO_HOME/$LEVELDB_DIR/$DATA_DIR/MANIFEST-*`
  MANIFEST_MD5_POST_BACKUP=`md5sum $MANIFEST_FILE_POST_BACKUP`
  if [ "$MANIFEST_MD5" = "$MANIFEST_MD5_POST_BACKUP" ] && [ "$copy_success" = "0" ]; then
    backup_succeeded=true
    break;
  fi
  attempts=$((attempts - 1))
  echo `date`" $0: leveldb manifest changed during backup process! $attempts attempts remaining."
done
```

```

if [ "$backup_succeeded" = false ]; then
    echo `date`" $0: Failed to backup with a consistent leveldb manifest!"
else
    echo `date`" $0: Backup created and verified leveldb manifest consistency!"
fi

if [ "$BACKUP_EXACT" = true ]; then
    rm -rf $BACKUP_TO/$LEVELDB_DIR"-unchanged"
    cp -r $BACKUP_TO/$LEVELDB_DIR"-tmp" $BACKUP_TO/$LEVELDB_DIR"-unchanged"
fi

backup_repaired=false
# Verify and repair using python script
python verify_leveldb.py $BACKUP_TO/$LEVELDB_DIR"-tmp"/$DATA_DIR
if [ "$?" != "0" ]; then
    echo `date`" $0: Discovered backup corruption! Attempting to repair!"
    python repair_leveldb.py $BACKUP_TO/$LEVELDB_DIR"-tmp"/$DATA_DIR
    if [ "$?" != "0" ]; then
        echo `date`" $0: Backup repair failed!"
    else
        python verify_leveldb.py $BACKUP_TO/$LEVELDB_DIR"-tmp"/$DATA_DIR
        if [ "$?" != "0" ]; then
            echo `date`" $0: Backup repair failed!"
        else
            echo `date`" $0: Backup repair succeeded!"
            backup_repaired=true
            backup_succeeded=true
        fi
    fi
fi
else
    echo `date`" $0: Backup passed corruption verification!"
fi

if [ "$backup_succeeded" = true ]; then
    rm -rf $BACKUP_TO/$LEVELDB_DIR
    if [ "$BACKUP_EXACT" = true ] && [ "$backup_repaired" = false ]; then
        mv $BACKUP_TO/$LEVELDB_DIR"-unchanged" $BACKUP_TO/$LEVELDB_DIR
    else
        mv $BACKUP_TO/$LEVELDB_DIR"-tmp" $BACKUP_TO/$LEVELDB_DIR
    fi
fi

```

The following script can be used to verify a leveldb cache for corruptions:

verify_leveldb.py

```
#!/usr/bin/python
import sys
import leveldb
from datetime import datetime
import subprocess
if (len(sys.argv) != 2) :
    print 'Usage:'
    print 'python verify_leveldb.py /path/to/leveldb/cache/'
    sys.exit(1)
dbpath = sys.argv[1]
try:
    startTime = datetime.now()
    print sys.argv[0], ": Inspecting db: ", dbpath
    db = leveldb.LevelDB(dbpath, create_if_missing=False, paranoid_checks=True)
    it = db.RangeIter(None, None, True, True, True)
    records = 0;
    for key, value in it:
        records = records + 1
    print sys.argv[0], ": Backup verification successful!"
    print sys.argv[0], ": Total records: ", records
    print sys.argv[0], ": Time taken:", datetime.now() - startTime
except:
    print sys.argv[0], ": Backup verification failed!"
    print sys.argv[0], ": Unexpected error:", sys.exc_info()[0]
    raise
    exit(1)
# Rename ldb files to sst (Python API uses ldb extension, whereas infinispn expects sst)
script = 'for f in ' + dbpath + '/*.ldb; do mv $f "${f%.ldb}.sst"; done'
subprocess.call(['/bin/bash', '-c', script])
```

Repairing Corrupt LevelDB

When the LevelDB database becomes corrupted, the RepairDB option provided by the LevelDB API can be used to recover as much as data as possible. In LevelDB, the manifest file holds account of all files and their corresponding key ranges. The recovery process inspects each file in the leveldb directory and updates the manifest accordingly. **This implies that even with a successful repair missing-files could lead to loss of data, which in turn can prevent a successful restoration of the repository.**

The below script can be used to repair corrupt leveldb cache:

repair_leveldb.py

```
#!/usr/bin/python
import leveldb
import sys
import os
import subprocess

if (len(sys.argv) != 2) :
    print 'Usage:'
    print 'python repair_leveldb.py /path/to/leveldb/cache'
    sys.exit(1)

path = sys.argv[1]
if not os.path.isdir(path) :
    print sys.argv[0], ": Directory %s does not exist!" % path
    sys.exit()

# Use the leveldb module to repair the files
leveldb.RepairDB(path)

# Rename ldb files to sst (Python API uses ldb extension, whereas infinispn expects sst)
script = 'for f in ' + path + '/*.ldb; do mv $f "${f%.ldb}.sst"; done'
subprocess.call(['/bin/bash', '-c', script])
```

